Memory with Memory: Soft Assignment in Genetic Programming

Nicholas Freitag McPhee
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota USA
mcphee@morris.umn.edu

Riccardo Poli
Department of Computing and Electronic Systems
University of Essex
Colchester, UK
rpoli@essex.ac.uk

ABSTRACT

Based in part on observations about the incremental nature of most state changes in biological systems, we introduce the idea of Memory with Memory in Genetic Programming (GP), where we use "soft" assignments to registers instead of the "hard" assignments used in most computer science (including traditional GP). Instead of having the new value completely overwrite the old value of the register, these soft assignments combine the old and new values.

We then report on extensive empirical tests (a total of 12,800 runs) on symbolic regression problems where Memory with Memory GP almost always does as well as traditional GP, while significantly outperforming it in several cases. Memory with Memory GP also tends to be far more consistent, having much less variation in its best-of-run fitnesses than traditional GP. The data suggest that Memory with Memory GP works by successively refining an approximate solution to the target problem. This means it can continue to improve (if slowly) over time, but that it is less likely to get the sort of exact solution that one might find with traditional GP. The use of soft assignment also means that Memory with Memory GP is much less likely to have truly ineffective code, but the action of successive refinement of approximations means that the average program size is often larger than with traditional GP.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*; I.2.6 [**Artificial Intelligence**]: Learning—*induction*

General Terms

Algorithms

Keywords

Genetic Programming, Linear GP, Soft assignment, Memory with memory, Symbolic regression

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-130-9/08/07...$5.00.

1. INTRODUCTION

In the vast majority of programming models, dating back to the Turing machine [17] and the earliest electronic computer architectures (e.g., [18]), assignments are entirely destructive in the sense that an instruction of the form `x:=y` or `LOAD R2 R1` completely overwrites the previous value of a memory location or register. That earlier value is lost forever, and has no impact on the future behavior of the system (unless it was copied elsewhere before it was overwritten). This overwriting model of assignment was carried over to most versions of genetic programming (GP) that had state and assignments. This includes linear GP [2], which evolves sequences of (virtual or real) machine code instructions that usually act by destructively writing to registers or memory locations.

This is in contrast to most biological systems, where the state of such a system is rarely if ever completely replaced with a new state with no regard for or "memory" of the previous state. Changes in protein concentrations in the cell, for example, can happen quickly, but are still typically incremental in nature, with each new state being constructed via modification of the previous state rather than complete replacement of it. Even dramatic state changes such as the transformation of a caterpillar into a butterfly take time and involve large numbers of small, local changes.

This difference with biology might be sufficient reason on its own to explore other models of assignment in GP. There are, however, practical concerns that also suggest that there might be value in alternative approaches. Linear GP systems with hard assignments, for example, can be quite fragile with respect to certain changes. A program that works by incrementally building up a result in a register can have its behavior radically altered by something like a point mutation that writes a 0 to the accumulating register late in the process. These hard assignments can also act as powerful intron creation mechanisms, as overwriting registers can render whole sequences of preceding instructions irrelevant to the final output of the evolved program.

Here we propose an alternative *Memory with Memory* model, with "soft" assignments that merge new values with previous values instead of overwriting them. We show that including this new type of assignment in a linear GP system can significantly improve performance on a variety of symbolic regression problems. We also find that Memory with Memory changes the nature of bloat, reducing the amount of ineffective code while at the same time tending to increase the mean program sizes as GP continues to refine the evolved approximations.

We begin by reviewing related work in the next section. We then define our linear GP system and our particular approach to Memory with Memory in Section 3. Our empirical results are presented in Section 4. Finally, we summarize our findings in Section 5.

2. RELATED IDEAS

While many GP systems are expression based, numerous GP systems have used some kind of memory or state. An obvious instance is linear GP [2], but other examples include indexed memory [16, 4, 1], and work on evolving data structures (such as stacks) that have internal state [9, 5]. Similarly, systems such as PushGP [15, 13] that use (rather than evolve) data structures such as stacks in their computational model are manipulating internal state in an important way. We are unaware, however, of any memory or (internal) state-based GP system that uses a soft assignment of the type proposed here.

Probably more similar to this work are systems where evolved agents rely primarily on external state (known as stigmergy). Here the state of an individual is rarely if ever completely replaced with a new state with no regard to the previous one. In swarm intelligence [3] and ant colony optimization [6], for example, changes in state such as pheromone levels are typically incremental, basing the new levels on the old. Similarly, the position of agents in particle swarm optimization systems [11] are almost always adjustments of the previous position rather than arbitrary jumps. This connection would also hold for many traditional GP systems used to evolve agents whose behavior is driven primarily by external state. In the classic artificial ant problem using the function set given in [8], for example, the new state of the ant following the trail is always a minor alteration of the previous state as the ant either turns 90° or moves forward one square. Similarly, many systems that evolve strategies for games like RoboCup Soccer [10] rely largely or entirely on external state which updates in an incremental fashion. [14] extends the idea of indexed memory by allowing multiple individuals in the population to read from and write to a shared memory space, which allows them to have both internal and external (shared) state; the write instructions in this system were traditional hard assignments, however, so the state was not constrained to incremental changes.

3. LINEAR GP WITH SOFT ASSIGNMENT

In this section we describe our basic linear GP system (Section 3.1), the details of our soft assignment mechanism (Section 3.2), the target polynomials we used in our experiments (Section 3.3), and how we computed the fitness of our evolved individuals (Section 3.4).

3.1 Linear GP

In this work we use a simple, register based linear GP system similar to that described in [2]. Table 1 lists the parameter settings used for our experiments; these values have proven useful in prior work, and no effort was made to optimize them for any of the systems used here.

Parameter	*Value*
Independent runs	50
Max initial length	50
Max length after XO	500
Point mutation rate (per primitive)	$1/\ell$
Population size	1000
Generations	40
Fitness evaluations per run (pop. size × gens)	40,000
Crossover rate (per individual)	0.9
Mutation rate (per individual)	0.1
Tournament size	2
Assignment hardness (γ)	1.0, 0.7, 0.5, 0.3

Table 1: Parameter settings used in these experiments (see Section 3.2 for further explanations on γ). These are fairly generic parameter choices and were not optimized for any of the systems used in these experiments.

The evolved programs are variable length linear sequences of "machine instructions" acting on a set of up to six registers (`R1-R6`); the full instruction set is given in Table 2. Note that when using traditional, hard assignment, `R3-R6` can only be used to store temporary results; all arithmetic is performed using just `R1` and `R2`. When using soft assignment, however, instructions like `R4:=R1` implicitly perform whatever calculations are used to implement soft assignments.

While there has been some study of function selection in GP (e.g., [7]), this remains more art than science. No particular attempt was made to fine tune our instruction set for the target functions used in this study. To see how soft assignment would perform with different instructions, however, we did use four progressively larger instruction sets F1 (group A from Table 2), F2 (groups A and B), F3 (groups A, B, C), and F4 (groups A, B, C, D). F1 is a basic two register system with just addition, multiplication, and swap. F2 adds the constants 0 and 1, subtraction and division, instructions to copy data to/from `R1` and `R2`, and swaps with `R3`. F3 adds hard assignment from the input variable x and the constants 0 and 1. F4 adds hard and soft assignment of $-x$ and -1 to `R1` and `R2`.

Our linear GP system uses a steady state control strategy with binary tournament selection for choosing parents and negative binary tournament selection for replacement. The initial population is generated by repeatedly creating random individuals with their length chosen uniformly from the range [1, 50].

New individuals are constructed using mutation 10% of the time, and subtree crossover 90% of the time; we use no reproduction. If mutation is chosen, a parent individual is selected via tournament selection, and an offspring is generated using either point mutation or subtree mutation with equal probability. With point mutation each instruction has a $1/\ell$ chance (where ℓ is the length of the program) of being replaced by a randomly selected instruction. With subtree mutation, a random point is chosen and all the instructions after that point are replaced by a new randomly generated sequence of instructions of length between 1 and 500 (the maximum allowed length after crossover).

If crossover is chosen, we select two parents (again, via tournament selection) and apply homologous two-point crossover with 50% probability, and subtree crossover with 50% probability. Subtree (or variable length) crossover involves the selection of one crossover point in each parent, and swapping the instructions following the crossover points. Since the crossover points are chosen independently, the length of the swapped suffixes can be different, leading to offspring of varying length. Homologous crossover [12, Section 7.1.3] requires choosing two crossover points which are used to divide both parents into three sections. The offspring is then formed by swapping the "middle" sections of the two parents, which means that the offspring is guaranteed to have the same length as the parent donating the prefix and suffix portions.

3.2 Memory with Memory: Soft assignment

There are numerous approaches that could be taken to combining the old values with the new when performing assignments. In this research we take a simple but flexible approach: weighted averaging of the old value of a register with the new value being assigned. In particular, if v_{old} is the original value of the register, and v_{new} is the new value being assigned to the register, the resulting value

Instruction sets
$F1 = A$
$F2 = A \cup B$
$F3 = A \cup B \cup C$
$F4 = A \cup B \cup C \cup D$

Group	*Description*	*Instructions*
A	Read input	`R1:=X`, `R2:=X`
	Plus, times	`R1:=R1+R2`, `R2:=R1+R2`, `R1:=R1*R2`, `R2:=R1*R2`
	Accum. swap	`Swap R1 R2`
B	Constants	`R1:=0`, `R2:=0`, `R1:=1`, `R2:=1`
	Minus, divides	`R1:=R1-R2`, `R2:=R1-R2`, `R1:=R1%R2`, `R2:=R1%R2`
	Copy to R1	`R1:=R2`, `R1:=R3`, `R1:=R4`, `R1:=R5`, `R1:=R6`
	Copy to R2	`R2:=R1`, `R2:=R3`, `R2:=R4`, `R2:=R5`, `R2:=R6`
	Copy from R1	`R2:=R1`, `R3:=R1`, `R4:=R1`, `R5:=R1`, `R6:=R1`
	Copy from R2	`R1:=R2`, `R3:=R2`, `R4:=R2`, `R5:=R2`, `R6:=R2`
	Swap R3	`Swap R1 R3`, `Swap R2 R3`
C	Hard input read	`R1:=X(H)`, `R2:=X(H)`
	Hard constants	`R1:=0(H)`, `R2:=0(H)`, `R1:=1(H)`, `R2:=1(H)`
D	Negative input read	`R1:=-X`, `R2:=-X`, `R1:=-X(H)`, `R2:=-X(H)`
	Negative one	`R1:=-1`, `R2:=-1`, `R1:=-1(H)`, `R2:=-1(H)`

Table 2: Linear GP instructions used in these experiments. All assignments are soft except those suffixed with "(H)". % is protected division, which returns its first argument if the second is less that 10^{-6}. The instructions are divided into four groups of related instructions (A, B, C, and D, indicated by horizontal lines). We used four progressively larger sets of instructions: F1 (group A), F2 (groups A and B), F3 (groups A, B, C), and F4 (groups A, B, C, D).

v_{result} is given by

$$v_{\text{result}} = \gamma v_{\text{new}} + (1-\gamma) v_{\text{old}} \tag{1}$$

where γ is a constant that indicates the *assignment hardness*, allowing us to determine how "hard" or "soft" the assignment operator is. If $\gamma = 1$, for example, we get a completely "hard" assignment, and have traditional GP. For $\gamma = 0.5$, on the other hand, we have a simple average of the new and old values. For $\gamma = 0$, instead, all registers are effectively write-protected, making all instructions behave like no-ops. In this situation all programs compute the identity function.

While having the advantages of simplicity and flexibility, this does introduce yet another parameter into the GP system. We make no attempt at comprehensively optimizing that new parameter here, but do perform extensive tests using a set of four values of γ: 1.0, 0.7, 0.5, and 0.3.

3.3 Target polynomials

We applied this new system to several classes of polynomial symbolic regression problems, using target polynomials of several different degrees for each class. In total we will test each value of γ and each instruction set on a total of 16 different symbolic regression problems of three different types, with degrees ranging from 4 to 15 for each of these three classes.

The simplest class of problems we explored were "regular" polynomials of the form

$$\sum_{1 \le n \le d} x^n = x + x^2 + x^3 + \cdots + x^d \tag{2}$$

We used six target polynomials of this form, having degrees $d = 5$, 7, 9, 11, 13, and 15; two examples are listed in Table 3. Low-degree polynomials of this type have commonly been used in GP studies [8, 12].

$$x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 \tag{3}$$

$$x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{12} + x^{13} \tag{4}$$

Table 3: Examples of two regular polynomials (see Equation (2)), having degrees 7 and 13 respectively.

A slightly more complex class of problems had coefficients that were randomly generated from the set $\{0, 1\}$.[1] Thus they are essentially the same as the "regular" polynomials, but with random terms removed. We used five such polynomials, with degrees 4, 7, 9, 12, and 15; the specific polynomials are listed in Table 4.

$$x + x^2 + x^3 + x^4 \tag{5}$$

$$x + x^2 + x^3 + x^4 + x^6 + x^7 \tag{6}$$

$$x + x^2 + x^3 + x^4 + x^6 + x^7 + x^8 + x^9 \tag{7}$$

$$x + x^2 + x^3 + x^4 + x^6 + x^7 + x^8 + x^9 + x^{12} \tag{8}$$

$$x + x^2 + x^3 + x^4 + x^6 + x^7 + x^8 + x^9 + x^{12} + x^{15} \tag{9}$$

Table 4: Polynomials generated using randomly chosen coefficients from the set $\{0, 1\}$.

Finally, we created a third class of polynomials by adding -1 in the set of possible coefficients; these are listed in Table 5.

Note that in each class the polynomials of higher degrees are "extensions" of the lower degree polynomials in the sense that the

$$1 - x + x^2 + x^3 - x^5 \tag{10}$$

$$1 - x + x^2 + x^3 - x^5 + x^8 - x^9 \tag{11}$$

$$1 - x + x^2 + x^3 - x^5 + x^8 - x^9 + x^{10} + x^{11} \tag{12}$$

$$1 - x + x^2 + x^3 - x^5 + x^8 - x^9 + x^{10} + x^{11} - x^{12} \tag{13}$$

$$1 - x + x^2 + x^3 - x^5 + x^8 - x^9 + x^{10} + x^{11} - x^{12} + x^{15} \tag{14}$$

Table 5: Polynomials generated using randomly chosen coefficients from the set $\{-1, 0, 1\}$.

[1] All random coefficients used in our test polynomials were generated via `http://www.random.org`.

γ	Estimated success rate	Lower	Upper
1.0	8.148%	7.686%	8.635%
0.7	10.578%	10.057%	11.122%
0.5	9.429%	8.935%	9.948%
0.3	7.679%	7.231%	8.153%

Table 6: Estimates of the success rates for each of the four assignment hardnesses using Wilson's method for computing (95%) confidence intervals for binomial probabilities. We define a success to be a run where the best fitness was less than 1.05, or an average error of less than 0.05 for each of the 21 test cases.

higher degree polynomials are equal to the lower degree polynomials plus some new higher order terms. For example, the degree 7 polynomial (Equation (6)) from Table 4 is the degree 4 polynomial (Equation (5)) with two additional terms ($x^6 + x^7$).

3.4 Evaluating fitness

For each target polynomial the fitness is the sum of the absolute error of the evolved function on 21 evenly spaced test points in the range [-1, 1]: $\{-1.0, -0.9, -0.8, \ldots, 0.8, 0.9, 1.0\}$. The target then is to minimize this error. We define a *success* to be a run where the best fitness was less than 1.05, or an average error of less than 0.05 over the 21 test cases.

4. RESULTS

To better understand what impact Memory with Memory and the particular value of γ has on symbolic regression problems, we performed 50 independent runs for each combination of

- 4 values of γ (1.0, 0.7, 0.5, 0.3)
- 4 instruction sets (F1-F4 from Table 2)
- 16 different polynomials (Section 3.3)

leading to a total of 12,800 runs and 512,000,000 fitness evaluations.

4.1 Impact of soft assignment on fitness

Figure 1 shows the distribution of the best fitness values from each of those 12,800 runs for each of the four values of the assignment hardness γ. $\gamma = 0.7$ has the best results of the four, with $\gamma = 0.5$ second, and $\gamma = 1$ (traditional GP) and $\gamma = 0.3$ being statistically indistinguishable.[2] The plot also indicates that the variance for the assignment hardnesses less than 1 are significantly smaller than for traditional GP, meaning that using soft assignment gives us more consistent results.

This is supported by the estimated success rates listed in Table 6. Both $\gamma = 0.7$ and $\gamma = 0.5$ have significantly better success rates than traditional GP ($\gamma = 1.0$). $\gamma = 0.7$, for example, has an estimated success rate of over 10% across all the combinations we ran, while the estimated success rate of traditional GP was just over 8%.

Figure 2 shows that the advantage of soft assignment is quite consistent across a wide range of degrees. The median for $\gamma = 0.7$ and $\gamma = 0.5$ are better than for $\gamma = 1$ in all cases except the degree 4 polynomial (Equation (5)) and the combination of the two degree 7 polynomials (Equations (3) and (6)) where they effectively tie. $\gamma = 0.7$ also does better than $\gamma = 0.5$ for several degrees, and

[2]Throughout this paper all tests for statistical significance are at 95% confidence levels.

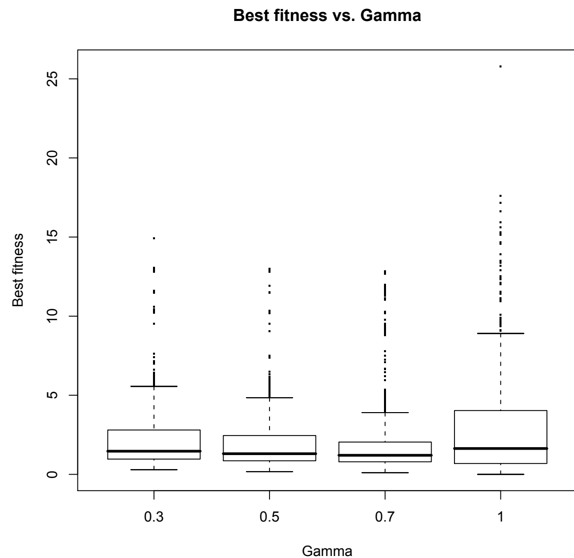

Figure 1: Boxplot showing the distribution of best fitnesses across all runs (all polynomial classes and degrees, all instruction sets). The differences in best fitnesses are statistically significant (using a pairwise Wilcoxon test) for all pairs except $\gamma = 0.3$ vs. $\gamma = 1$ (traditional GP). The differences in variances are also statistically significant (using the Fligner-Killeen test of homogeneity of variances).

never does worse. Note that the best (minimum) for traditional GP is consistently nearly perfect (very nearly 0), while the soft assignment runs were generally unable to reach a perfect solutions. This isn't surprising given that soft assignment tends to encourage refinement over time, which is likely to lead to approximate solutions. Those approximations are generally very close in our runs, but linear GP with soft assignment doesn't seem to have the "killer instinct" needed to finally reach the target, at least with the parameters used here.[3] This tendency to approximate may account for the advantage of hard assignment on the degree 4 polynomial; that polynomial is sufficiently easy that traditional GP solves it exactly with a high probability, while the Memory with Memory approximations are slightly worse. The variance trend noted earlier continues here, with the variance for traditional GP being similar in a few cases, but dramatically greater in others (e.g., degree 13 cases).

Figure 3 shows the proportion of successful runs by degree. Here again $\gamma = 0.7$ consistently does as well or better than all the other settings, and $\gamma = 0.3$ and traditional GP typically do the worst.

With one exception, all the instruction sets F1-F4 had at least moderate levels of success on the different polynomial classes. As can be seen in Figure 4, the instruction set F1 was generally incapable of solving any of the polynomials in the {-1, 0, 1}-polynomial class.

Figure 5 plots the distribution of best fitnesses across the 40 generations for $\gamma = 0.7$ and traditional GP ($\gamma = 1$). To help clarify the pattern, we plotted the application of a single instruction set (F2) to a single polynomial (Equation (4), the regular polynomial of degree

[3]Given that we only used 40,000 fitness evaluations per run, it's entirely possible that more generations would allow our system to further refine the solutions, but we haven't explored the impact of either increasing the population size or the number of generations.

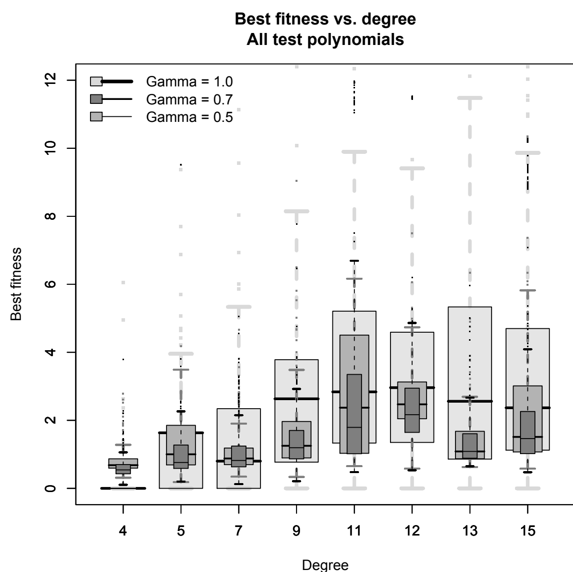

Figure 2: Boxplots of the best end-of-run fitness by degree for $\gamma \in \{1.0, 0.7, 0.5\}$. There are several outlier fitness values between 12 and 26 that are not visible in this plot.

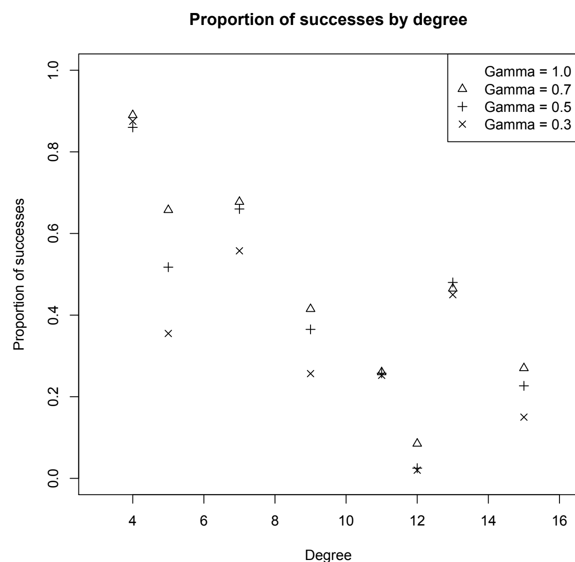

Figure 3: Proportion of successes by degree for each of the four values of γ. The proportion of successes for a configuration is the proportion of runs with that configuration that had a total error of less than 1.05.

13). We again see that the variance is much smaller with soft assignment. We also see that the fitness improves much more quickly in the early generations than traditional GP.

In Figure 6 we have the proportion of these F2-Degree 13 runs that showed any improvement from one generation to the next. $\gamma = 0.7$ shows a consistently higher proportion of improvements across all the generations than traditional GP ($\gamma = 1$). Both the hard and soft assignments showed a steady drop in the proportion of runs showing improvement up to around generation 20, where in both cases the slope flattens off somewhat. For $\gamma = 0.7$, however, the proportion of runs stays almost constant in the later generations, while it's continuing to drop noticeably for traditional GP. At generation 40 the proportion of runs still showing improvement in any given generation is over 30% for the soft assignment runs, while it's only around 10% for the traditional GP runs. It's important to realize that, however, most of these improvements in the later generations are quite small, as is suggested by Figure 5. Thus while we would expect continued improvements in best fitness if we let the soft assignment runs continue for additional generations, many of those improvements would be extremely small improvements in the evolved approximations.

4.2 Impact of soft assignment on length

Hard assignment is one potential source of ineffective or "intron" code in GP, i.e., sequences of instructions which do work, but ultimately have no impact on the program's final output. An instruction like `R1:=0`, for example, effectively undoes any preceding operations that collected results in `R1`, potentially making long sequences of previous instructions ineffective. This suggests that hard assignment could play a role in bloat [12, Section 11.3] by providing a mechanism for program lengths to increase without improving (or even changing) their functionality. With soft assignment, on the other hand, every instruction has some impact on the future state of the system. While an early instruction might have a very limited impact in a long program, it will have some impact, implying that there is ultimately no truly ineffective code when using soft assignment. The ability for GP to use soft assignment to incrementally approximate solutions, however, suggests that programs could grow longer and longer over time as GP works to improve its approximations.

Figure 7 shows the distribution of both mean and best-of-run program lengths for all four values of γ across all the runs. Both the mean and best-of-run program lengths were substantially larger for $\gamma = 0.7$ and $\gamma = 0.5$ than for traditional GP. The maximum size allowed after crossover (500) would push down the the amount the ineffective code one would expect in traditional GP. In the case of soft assignment, however, where all instructions have an impact and where there are steady, if small, improvements in fitness over time (as seen in Figures 5 and 6), there was presumably more fitness correlated pressure to grow.

Figure 8 shows that the tendency for soft assignment to lead to larger programs holds for all four function sets, and is much more pronounced for F3 and F4 than F1 and F2. The best-of-run sizes with F1 were substantially smaller than with the other three instruction sets for both traditional GP and $\gamma = 0.7$. The best-of-run sizes in fact showed statistically significant differences across all four function sets when using traditional GP. The best-of-run sizes were much more similar, however, when using $\gamma = 0.7$, especially for F2 and F3; in fact, the differences were not statistically significant for the pairs (F2, F3) and (F2, F4).

Table 7 shows the differences in the proportions of the different instructions (from set F2) as they appeared in the best-of-run individuals for the 50 runs with F2 on the regular degree 13 polynomial (Equation 4) between $\gamma = 0.7$ and traditional GP ($\gamma = 1$). Negative values at the top of the table (e.g., `R1:=X`, `R2:=X`, and `R1:=R1+R2`) all appear more frequently in the best-of-run individuals in the soft assignment runs. This indicates, for example, that runs with soft assignment used considerably higher proportions of reads from the input variable. In traditional GP, such a read would completely overwrite the contents of the register being read into, and would consequently need to be used with some caution. In our Memory

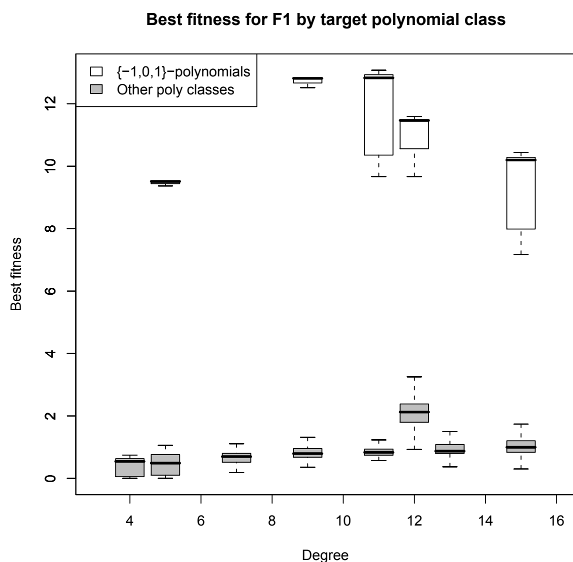

Figure 4: Distribution of end-of-run best fitnesses by class of target polynomial, restricted to runs using the instruction set F1. (All function sets had reasonable levels of success on all test classes except F1 on the {-1, 0, 1}-polynomials.) The white boxplots show the distribution for the {-1, 0, 1}-polynomials, while the gray boxplots show the aggregate results for all the polynomials in the other two classes.

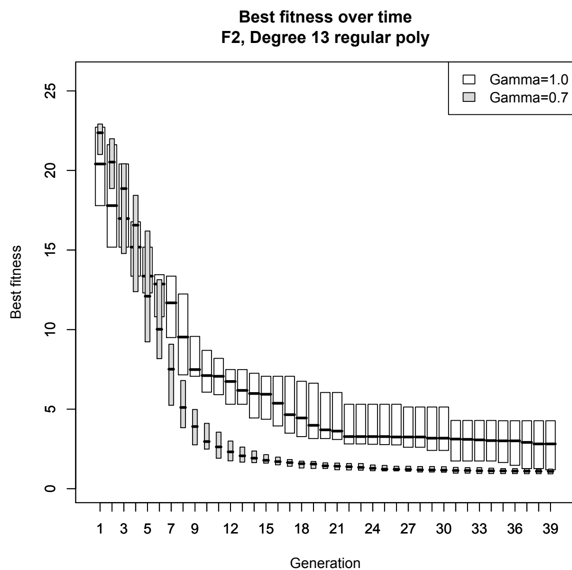

Figure 5: Boxplots showing the middle 50% of the data for the best fitness over time for $\gamma = 0.7$ and for traditional GP ($\gamma = 1$). These results are for a single instruction set (F2) and single polynomial (Equation (4), the regular polynomial of degree 13).

with Memory system, however, such an instruction blends the contents of a register with the input variable *x* (which is not in fact a terrible approximation of the target function). Conversely, traditional GP was more likely to include the division instructions (near the bottom of the table) as well as, to a lesser degree, subtraction and multiplication. Given that neither division or subtraction are needed to solve the problem, their increased presence in the traditional GP runs could help explain traditional GP's generally poorer performance on this problem.

5. CONCLUSIONS

In this paper we have introduced the idea of Memory with Memory GP, where we use "soft" assignments to registers instead of the "hard" assignments used in most computer science (including traditional GP). Instead of having the new value completely overwrite the old value of the register, these soft assignments combine the two values, using a weighted average in the work reported here.

Our extensive empirical tests (a total of 12,800 runs) with symbolic regression problems show that Memory with Memory GP almost always does as well as traditional GP, while significantly outperforming it in several cases. Memory with Memory GP also tends to be far more consistent, having much less variation in its best-of-run fitnesses than traditional GP. The data suggest that Memory with Memory GP works by successively refining an approximate solution to the target problem. This means it can continue to improve (if slowly) over time, but that it is less likely to get the sort of exact solution that one might find with traditional GP. The use of soft assignment also means that Memory with Memory GP is much less likely to have truly ineffective code, but the action of successive refinement of approximations means that the average program size is often larger than with traditional GP.

6. FUTURE WORK

This is a first exploration of a new approach to state updates in GP, and could only examine a handful of the many alternatives.

An obvious area for future exploration would be the specific implementation of soft assignment. We used a weighted average between the previous and new value (Section 3.2), but one could, for example, use moving averages, where only the last *k* values impact the new value, allowing old values to be completely "forgotten" over time. Other approaches such as non-linear combinations could certainly be explored. We also didn't perform a detailed exploration of different values of γ, or its interaction with other parameters of our evolutionary system.

Another issue not addressed here is whether it would be beneficial to distinguish between different kinds of assignments, making some soft and some hard. An instruction like `R1:=R1+R2`, for example, already incorporates the old value of `R1`, and it could be argued that soft assignment is unnecessary there. The instruction `R1:=R1*R2` also typically includes the old value of `R1`, but if `R2:=0` then the old value of `R1` is completely overwritten. Function sets F3 and F4 both contain hard and soft versions of some assignments, so we can get a sense of how evolution combines soft and hard assignment operators. A more sophisticated option would be to allow each instruction to have its own value of γ which could be adjusted over time via some process (e.g., evolution or backpropogation).

One of the challenges with soft assignment is that it makes the evolved solutions harder to represent and analyze since every assignment is in fact a linear combination of two values. An interesting possibility would be to start with $\gamma < 1$, but progressively move it to 1 over the course of a run. This might have the effect of smoothing the fitness landscape, but it's not clear how easily the system would transition from the approximations generated by soft assignment to a successful (exact) solution using hard assignments.

Similarly, one could start with hard assignments, but when a run appears stuck, decrease γ (making assignments softer) in the hope

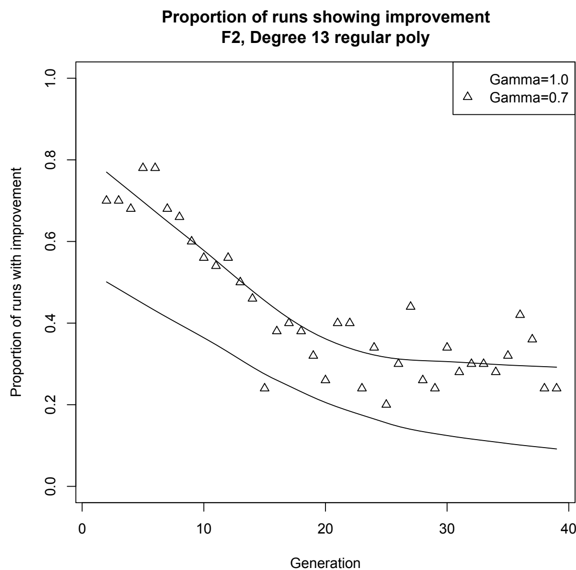

Figure 6: Proportion of runs showing improvement in best fitness over time for $\gamma = 0.7$ and for traditional GP ($\gamma = 1$). Each set of points is overlaid with a curve generated using local polynomial regression fitting (a lowess curve). These results are for a single instruction set (F2) and a single polynomial (Equation (4), the regular polynomial of degree 13).

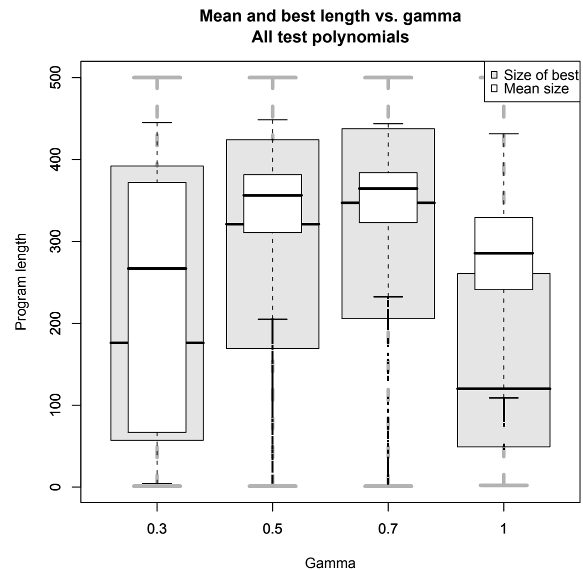

Figure 7: Boxplots showing the distribution of both mean and best-of-run program lengths for the four different values of gamma used in this study, across all runs (all polynomial classes and degrees, all instruction sets). The differences across the γ for both the mean and best-of-run lengths are statistically significant using pairwise Wilcoxon tests. The differences in variances across both the mean and best-of-run lengths are also statistically significant (using the Fligner-Killeen test of homogeneity of variances).

of introducing a gradient or at least a neutral network that would allow for additional progress, possibly increasing γ again when progress has resumed.

Finally, an interesting direction to take this work is to look at how it performs on noisy and dynamic problems, e.g., problems where the target function changes over time.

Acknowledgments

We would like to thank EPSRC (grant EP/G000484/1) for financial support, and Dagstuhl seminar 08051 on the Theory of Evolutionary Algorithms, where much of this paper was finalized. Nic would also like to thank Riccardo and the University of Essex for being such gracious hosts during his research sabbatical.

7. REFERENCES

[1] P. J. Angeline. An alternative to indexed memory for evolving programs with explicit state representations. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 423–430, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, Jan. 1998.

[3] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence : From Natural to Artificial Systems (Santa Fe Institute Studies on the Sciences of Complexity)*. Oxford University Press, USA, September 1999.

[4] S. Brave. Evolving recursive programs for tree search. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[5] W. S. Bruce. Automatic generation of object-oriented programs using genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 267–272, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[6] M. Dorigo and T. Stützle. *Ant Colony Optimization (Bradford Books)*. The MIT Press, July 2004.

[7] W. Gang and T. Soule. How to choose appropriate function sets for GP. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 198–207, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.

[8] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[9] W. B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 Apr. 1998.

[10] S. Luke. Genetic programming produced competitive soccer softbot teams for RoboCup97. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222, University of

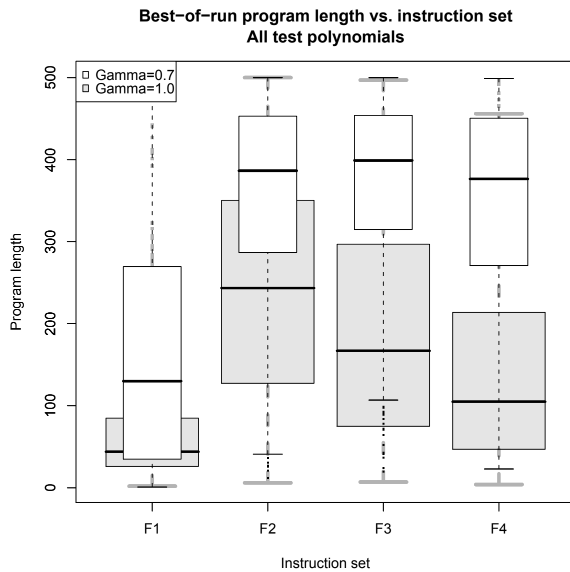

Figure 8: Boxplots showing the distribution of the best-of-run program lengths for the four different instruction sets F1-F4, for $\gamma = 1.0$ and $\gamma = 0.7$, across all polynomial classes and degrees. All differences in the best-of-run sizes for $\gamma = 1.0$ are statistically significant using a pairwise Wilcoxon test. For $\gamma = 0.7$, however, all pairwise differences were statistically significant (again using a pairwise Wilcoxon test) *except* the pairs (F2, F3) and (F2, F4).

Instruction	*Difference in proportions*
`R1 := X`	-0.025
`R2 := X`	-0.022
`R1 := R1 + R2`	-0.012
`R5 := R1`	-0.009
`R4 := R2`	-0.009
`R2 := 1`	-0.009
`R2 := R1 + R2`	-0.007
`R3 := R2`	-0.006
`R2 := R1`	-0.005
`R3 := R1`	-0.005
`R1 := R2`	-0.001
`R1 := 1`	0.000
`R6 := R1`	0.000
`Swap R1 R3`	0.000
`Swap R2 R3`	0.000
`R2 := R5`	0.000
`R2 := R6`	0.000
`R4 := R1`	0.001
`R1 := R1 * R2`	0.002
`R6 := R2`	0.002
`R1 := R3`	0.003
`R5 := R2`	0.004
`R1 := 0`	0.004
`Swap R1 R2`	0.004
`R1 := R4`	0.004
`R2 := R1 - R2`	0.005
`R2 := R3`	0.006
`R1 := R1 - R2`	0.006
`R2 := R4`	0.006
`R1 := R6`	0.007
`R2 := R1 * R2`	0.008
`R1 := R5`	0.009
`R2 := R1 % R2`	0.011
`R2 := 0`	0.012
`R1 := R1 % R2`	0.013

Table 7: The difference in proportions of the different instructions (from F2) as they appeared in the best-of-run individuals in the runs on the degree 13 regular polynomial (Equation (4)) using $\gamma = 0.7$ and traditional GP ($\gamma = 1.0$). Negative values mean that those instructions appeared with higher proportion in the soft assignment ($\gamma = 0.7$) runs, while positive values mean that those instructions appeared with a higher proportion in the traditional GP runs.

Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[11] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, June 2007.

[12] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[13] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[14] L. Spector and S. Luke. Cultural transmission of information in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 209–214, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[15] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[16] A. Teller. The evolution of mental models. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[17] A. M. Turing. *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma*, chapter "On Computable Numbers, with an Application to the Entscheidungsproblem", pages 58–87. Oxford University Press, 2004.

[18] J. von Neumann. First draft of a report on the EDVAC. Technical report, United States Army Ordnance Department and the University of Pennsylvania, 1945. Accessed at `http://www.virtualtravelog.net/entries/2003-08-TheFirstDraft.pdf`, 24 Jan 2008.